Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement

Hidetake Uwano, Masahide Nakamura, Akito Monden and Ken-ichi Matsumoto

Nara Institute of Science and Technology
{hideta-u, masa-n, akito-m, matumoto}@is.naist.jp

Abstract

This paper proposes to use eye movements to characterize the performance of individuals in reviewing source code of computer programs. We first present an integrated environment to measure and record the eye movements of the code reviewers. Based on the fixation data, the environment computes the line number of the source code that the reviewer is currently looking at. The environment can also record and play back how the eyes moved during the review process. We conducted an experiment to analyze 30 review processes (6 programs, 5 subjects) using the environment. As a result, we have identified a particular pattern, called scan, in the subjects' eye movements. Quantitative analysis showed that reviewers who did not spend enough time for the scan tend to take more time for finding defects.

Keywords: eye movement, source code review, human factor, computer program

CCS: H.1.2 [Models and Principles]: User/Machine Systems – Human factors; D.2.5 [Software Engineering]: Testing and Debugging – Code inspections and walk-throughs.

1 Introduction

Source code review (or simply code review) is peer review of source code of computer programs. It is intended to find and fix defects (i.e., bugs) overlooked in early development phases, improving overall code quality [Boehm 1981]. Basically, the code review is an off-line task conducted by human reviewers without compiling or executing the code. In the code review, a reviewer reads the code, understands the behavior, and detects and fixes defects if any. Especially in developing large-scale software applications, the code review of individual modules is vital, since it is quite expensive to fix the defects in later integration and testing stages. A study shows that review and its variants such as walk-through and inspection can discover 50 to 70 percent of defects in software product [Weigers 2002]. Our long-term goal is to establish an efficient method that allows the reviewer to find as many defects as possible.

Several methodologies that can be used for the code review have been proposed so far. The idea behind these methods is to pose a certain criteria on *reading* the documents. Code review without any reading criteria is called *Ad-Hoc Review* (AHR). A method where the reviewers read the code from several different viewpoints, such as designers, programmers and testers, is called *Perspective-Based Reading* (PBR) [Shull et al. 2000]. *Checklist-Based Reading* (CBR) [Fagan 1976] introduces a checklist with which the reviewers check typical mistakes in the code. *Usage-Based Reading* (UBR) [Thelin et al. 2001] is to review the code from users' viewpoint. *Defect-Based Reading* (DBR) [Porter et al. 1995] focuses on detecting specific type of defects.

To evaluate the performance of these methods, hundreds of empirical studies have been conducted [Ciolkowski et al. 2002]. However, there has been no significant conclusion on which review method is the best. Some empirical reports have shown that CBR, which is the most used methods in the software industries, is not more efficient than AHR. As for UBR, PBR, and DBR, they achieved slightly better performance than CBR and AHR [Basili et al. 1996; Porter et al. 1995; Porter and Votta 1998; Shull 1998; Thelin et al. 2003]. On the other hand, a study by Halling et al. [2001] reports an opposite observation that CBR is better than PBR. Several case studies have shown that these methods had no significant difference [Fusaro et al. 1997; Lanubile and Visaggio 2000; Miller et al. 1998; Sandahl et al. 1998].

The reason why the results vary among the empirical studies is that *the performance of individual subjects is more dominant than the review method itself*, since the review is a task involving many human factors. Thelin et al. [2004] compared the effectiveness, i.e., the defect detection ratio (Defects found / Total), between UBR and CBR. Fig. 1 depicts the result, showing that the effectiveness of UBR is 1.2 – 1.5 times better than the one of CBR on average. However, as seen in the dotted lines in the figure, the individual performance in the same review method varies much more than the method-wise difference. Unfortunately, the performance variance in individual reviewers has not been well studied. Thus, we consider it essential to investigate the performance of reviewers rather than to devise review methods. Hence, the key is how to capture the difference among good and bad reviewers.

To characterize the reviewer's performance in an objective way,

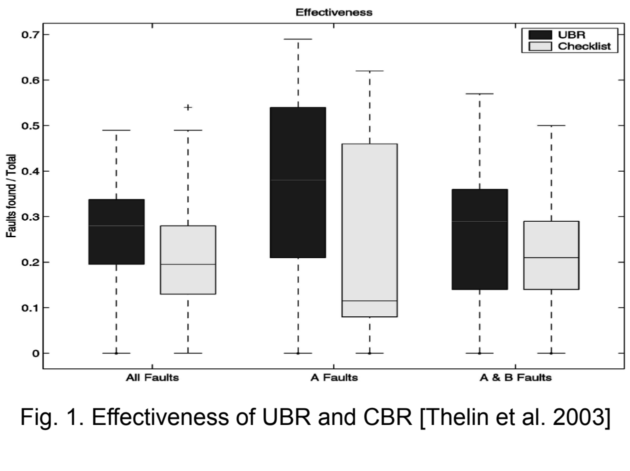


Fig. 1. Effectiveness of UBR and CBR [Thelin et al. 2003]

this paper proposes to use *eye movements* of the reviewer. In general, the source code is not read as ordinary documents like newspapers or stories. The reviewer frequently jumps from a line to another or compares multiple lines, to simulate the behavior of the program. The way of reading the code should vary among different reviewers, and the reading actions must appear on the eye movements of the reviewers. Therefore, we consider that the eye movements can be used as a powerful and objective metric to characterize the individual performance in the code review.

In this paper, we first present an integrated environment to measure and record the eye movements during the code review. We developed a software application, "Crescent", built on top of a non-contact eye mark tracker EMR-NC. Based on the raw data captured by the eye mark tracker, Crescent computes the line number of the source code that the reviewer is currently looking at. As an eye moves, Crescent records the transition from a line to another, as well as the duration time that the gaze stayed at every line. Also, Crescent has a feature that can play back the record automatically or interactively.

Using the environment, we then conduct an experiment of code review with five graduate students. Through the experiment, we have identified a particular pattern, called *scan*, in the subjects' eye movements. The scan pattern characterizes an action that the reviewer reads the *entire* code before investigating the details of each line. Quantitative analysis showed that reviewers who did not spend enough time for the scan tend to take more time for finding defects. Thus, it is expected that the eye movements is promising to establish a more human-centered code review method reflecting human factors.

The rest of this paper is structured as follows. Section 2 discusses related research in eye movement applications, and Section 3 explains the integrated environment. In Sections 4 and 5, we discuss the experiment and analysis. Finally, Section 6 concludes the paper with future work.

2 Related Work

Eye movements have been often used for the purpose of evaluating human performance, especially in cognitive science. Law et al. [2004] analyzed eye movements of expert and novice in laparoscopic surgery training environment. This study showed that experts tend to see affected parts more than the tool in hand, compared with novices. Kasarskis et al. [2001] investigated eye movements of pilots in a flight simulator. In this study, novices tend to concentrate seeing the altimeter than experts, while the experts see the airspeed.

Also, in the field of software engineering, there are several research works exploiting the eye movements, for the purpose of, for instance, monitoring on-line debugging processes [Stein et al. 2004; Torii et al. 1999], usability evaluation [Bojko and Stephenson 2005; Nakamichi et al. 2003], human interface [Jacob 1995; Zhai et al. 1999], and program comprehension[Crosby and Stelovsky 1990].

However, as far as we know, there is no research using the eye movements for analyzing individual performance in the source code review.

3 Capturing Eye Movements in Code Review

3.1 Requirements

We first discuss what should be required for measuring eye movements in the context of code review, according to specific characteristic of the review task. The requirements make it clear the purpose of the target measurement environment.

Requirement R1: Line-wise tracking of eye movements

A primary construct of a program is a *statement* and most programs are written in one-statement-per-line basis. So, it is reasonable to consider that the reviewer reads the code in units of *lines*. Hence, the measuring environment has to be capable of identifying which *line* of the code the reviewer is currently looking at. Note that the information must be stored as *logical* line numbers, which is independent of the font size or the absolute position where the code lines are currently displayed.

Requirement R2: Identification of reviewer's focus

Even if an eye mark comes at a line in the code, it does not necessarily mean that the reviewer is reading the line. That is, the measuring environment has to be able to distinguish a *gaze* from a *glance*. It is reasonable to assume that the reviewer has focused a line if the eye mark stayed in the line for a period.

Requirement R3: Record of time-sequenced lines

The order in which the reviewer reads lines is important information to reflect individual characteristics of code review. Also, each time the reviewer gazes at a line, it is essential to measure *how long* the line is being focused. The duration of the focus may indicate strength of reviewer's attention to the line. Therefore, the measurement environment must record the focused lines as *time sequence* data.

Requirement R4: Analysis supports

Preferably, the measuring environment should provide tool supports to facilitate analysis of the recorded data. Especially, features to play back and visualize the data significantly contribute to efficient analysis. The tools may be useful for subsequent interviews or for educational purposes to novice reviewers.

3.2 Integrated Measuring Environment

Based on the above requirements, we have developed an integrated environment for measuring eye movements in code review. The environment consists of hardware components including a non-contact eye camera, and a software application Crescent to process the tasks specific to code review. In the environment, the reviewer reads the code on a PC monitor.

Hardware components

In order to track the eye movement on each line (see Requirement R1), high resolution and precision are required for the eye camera. Therefore, we selected a non-contact eye mark tracker EMR-NC, manufactured by NAC Image Technology Inc (http://www.nacinc.jp/). EMR-NC can sample the eye movements within 30Hz. The finest resolution of the tracker is 5.4 pixels on the screen, which is equivalent to 0.25 lines of 20 point letters.

To display the source code, we used a 21-inches liquid crystal display (EIZO FlexScanL771) set at 1024x768 resolutions with a dot pitch of 0.3893 millimeter. To minimize the noise data, we prepared a fixed and non-adjustable chair for the reviewers.

The data sampled by EMR-NC is polled to a PC through RS-232C interface. On the PC, an application bundled with EMR-NC stores the data in a file with the CSV format. Each sample of the data consists of an *absolute* coordinates of the eye mark on the screen, and sampled date. The bundled application also can compute *fixations*, particular coordinates at which the eye mark stays for a given moment. The fixations can be useful to identify the reviewer's gaze (Requirement R2). Note however that the raw sampled data is not sufficient to satisfy Requirements 1 to 4, since it just represents a set of absolute coordinates of the eye movements. We need to refine the raw data into the one feasible to the individual analysis of code review.

Crescent: A software application manipulating gaze data

To complete the requirements, we have developed a software application Crescent (Code Review Evaluation System by Capturing Eye movemeNT) on top of the hardware components. Crescent manipulates the sample data polled from the eye camera, and converts the data to the line-wise information feasible to individual analysis of code review. Crescent was developed in the Java language with SWT (Standard Widget Tool), comprising about 4000 lines of code. Fig. 2 shows the system structure and the measurement environment. Using the figure, we explain how Crescent is designed to meet Requirements R1 through R4 as follows.

What most technically difficult is to satisfy Requirement R1. We need to convert the absolute coordinates sampled in pixels by the eye tracker, into logical line information. When Crescent is launched, the *output* module pops up a main textbox displaying a source code to be reviewed (see Fig. 3). It then opens the raw data file and the fixation data file to get the eye movement information. Based on the absolute position of the textbox and the absolute coordinates of eye mark, the *converter* module computes the *relative* coordinates of the eye mark in the textbox. Next, taking the font size, the line pitch and the screen resolution into account, the module converts each relative coordinate into a line number in the textbox which the coordinate corresponds to. Note that the reviewer may *scroll down* (or *up*) the source code using a slider bar besides the textbox, which changes the correspondence between the logical line number and the absolute position. For this, the *scrolling capture* module monitors all events of the slider bar, and adapts the correspondence to maintain the consistency of the line number.

To satisfy Requirement R2, we extensively use fixations obtained by the bundled application of EMR-NC. For given *fixation criteria* (i.e., *pixels* in diameter and *staying time* of eye mark), the application extracts the absolute coordinates of the fixations from

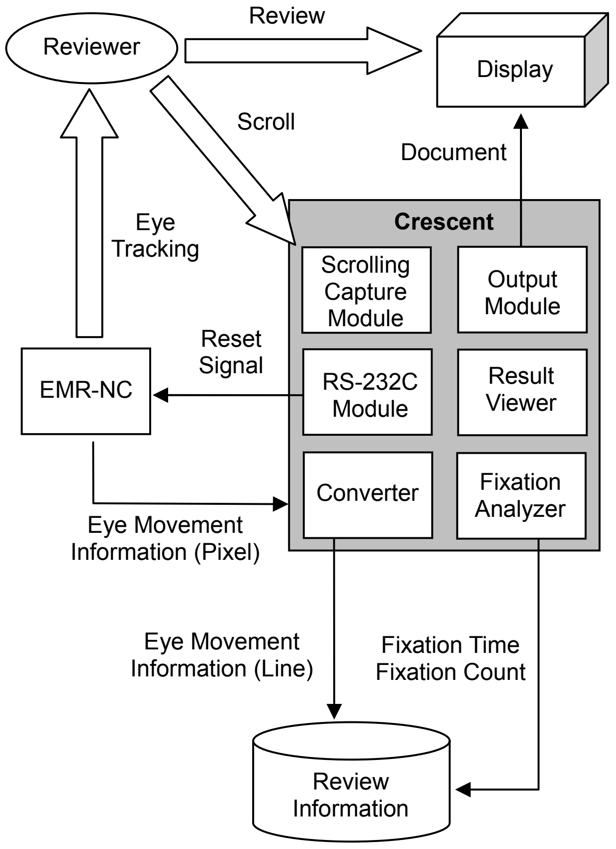
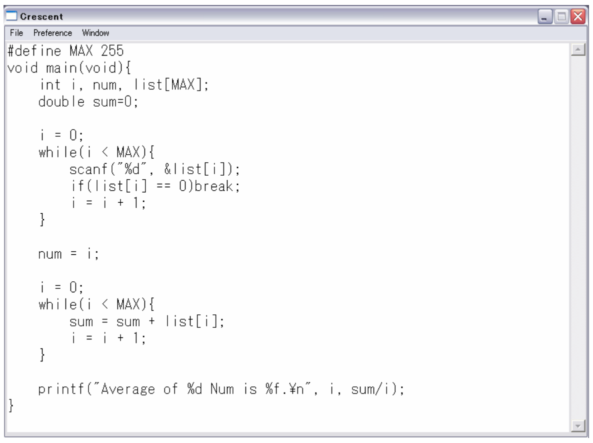
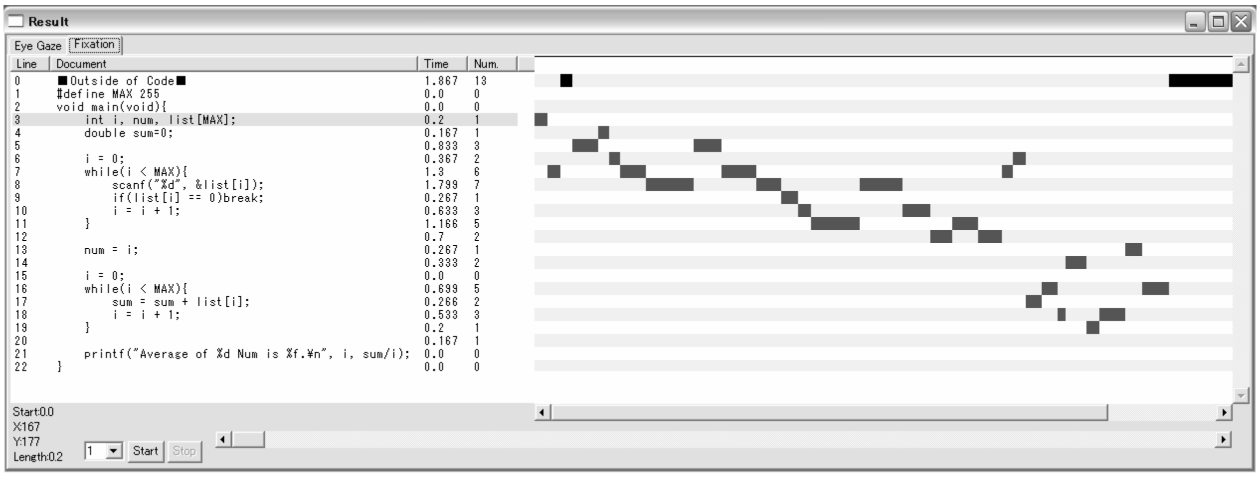


Fig. 2. Measurement environment of Crescent



```
#define MAX 255
void main(void){
    int i, num, list[MAX];
    double sum=0;

    i = 0;
    while(i < MAX){
        scanf("%d", &list[i]);
        if(list[i] == 0)break;
        i = i + 1;
    }

    num = i;

    i = 0;
    while(i < MAX){
        sum = sum + list[i];
        i = i + 1;
    }

    printf("Average of %d Num is %f.¥n", i, sum/i);
}
```

Fig. 3. Textbox for code review



Line	Document	Time	Num.
0	■Outside of Code■	1.867	13
1	#define MAX 255	0.0	0
2	void main(void){	0.0	0
3	int i, num, list[MAX];	0.2	1
4	double sum=0;	0.167	1
5		0.833	3
6	i = 0;	0.367	2
7	while(i < MAX){	1.3	6
8	scanf("%d", &list[i]);	1.799	7
9	if(list[i] == 0)break;	0.267	1
10	i = i + 1;	0.633	3
11	}	1.166	5
12		0.7	2
13	num = i;	0.267	1
14		0.333	2
15	i = 0;	0.0	0
16	while(i < MAX){	0.699	5
17	sum = sum + list[i];	0.266	2
18	i = i + 1;	0.533	3
19	}	0.2	1
20		0.167	1
21	printf("Average of %d Num is %f.¥n", i, sum/i);	0.0	0
22	}	0.0	0

```
Start:0.0
X:167
Y:177
Length:0.2
```

Fig. 4. Result viewer

the raw data sampled. The absolute coordinates are converted into the logical line number as explained above, which identifies a set of lines the reviewer focused on.

The *fixation analyzer* module summarizes the focused lines as a time-sequenced data, by investigates the fixations and the date information of each eye mark. Also, for each line the fixation analyzer measures the total time that the reviewer has focused, and the number of focuses, which completes Requirement R3.

Finally, to cope with Requirement R4, Crescent is equipped with a *result viewer*, which can play back and visualize the recorded line information. Fig. 4 shows a snapshot of the result viewer. The viewer displays the source code reviewed and a horizontal bar chart showing the line-wise eye movements of the reviewer. The sequence of the movements can be played back by highlighting the focused line on the source code. It is possible to play back the record from any time lines of the measurements.

Crescent also has a feature to send a reset signal to the eye mark tracker, which is implemented by the RS-232C module. This is to synchronize the starting (or the completion) time of the code review process, with the beginning (or the ending) of the eye mark measurement. The synchronization minimizes the margin of error between date of eye mark and date of scrolling event.

4 Experimental Evaluation of Individual Performance of Code Review Using Eye Movements

4.1 Experiment

Using the constructed environment, we performed an observation experiment of code review process.

Preliminaries

Five graduate students participated in the experiment as the reviewers. They have 3 or 4 years experience of programming, and have experienced code review before at least once.

We have prepared 6 small-scale programs written in the C language, each of which is comprised of 12 to 23 lines of source code. To measure the individual performance purely with the eye movement, we omitted any comments from the source code. We also prepared a specification for each program. The specification is compact and easy enough for the reviewer to understand and memorize. Then, in the source code of each program we intentionally injected a single defect. Every defect is logical defect, that is, no of syntax error is injected. We explain to reviewers each source code has only one logical defect.

Table 1 summarizes a list of the programs, the specifications and the defect injected. In this experiment, we determined the fixation criteria as the area of 30 pixels in diameter where the eye mark stays more than 50ms.

Task of Code Review

We instructed individual subjects to conduct code review of the six programs, using the developed measuring environment. We use Latin square to assign order of presentation for minimize learning/fatigue effects. For each program, the source code and the specification of the program were given to the subject. We allowed the subjects to ask questions on the specifications and the C language (except about the injected defects), before and during the review. The review method was the ad-hoc review, that is, neither a checklist nor a perspective was given.

A task for each subject to review a single source code consists of the following steps.

1. Calibrate the eye tracker so that the eye movements of the subject are logged correctly.
2. Explain the specification of the program to the subject verbally.
3. Synchronizing the subject to start the code review to find defect, start the capture of eye movements and code scrolling.
4. Suspend the review task when the subject tells he/she found the defect. Then, ask the subject to explain the defect verbally.
5. Finish the code review task if the detected defect is correct. Otherwise, resume the task going back to the step 3. The review task is continued until the subject successfully finds the defect, or the total time for the review exceeds 5 minutes.

Validation of Captured Data

After the experiment, we checked the validity of sampled coordinates. The coordinates where the subject was blinking or seeing outside the textbox were discarded. If data for a task contained invalid coordinates more than 30 percent of whole samples, we discarded the data from the analysis. Out of total 30 review tasks (i.e., 6 programs x 5 subjects), three trials were discarded.

4.2 Analyzing Result

The eye movements during the code review were visualized and played back by the result viewer. For instance, Fig. 5 (a) depicts the eye movements captured when subject E reviewed the source

Table 1 Programs reviewed in the experiment

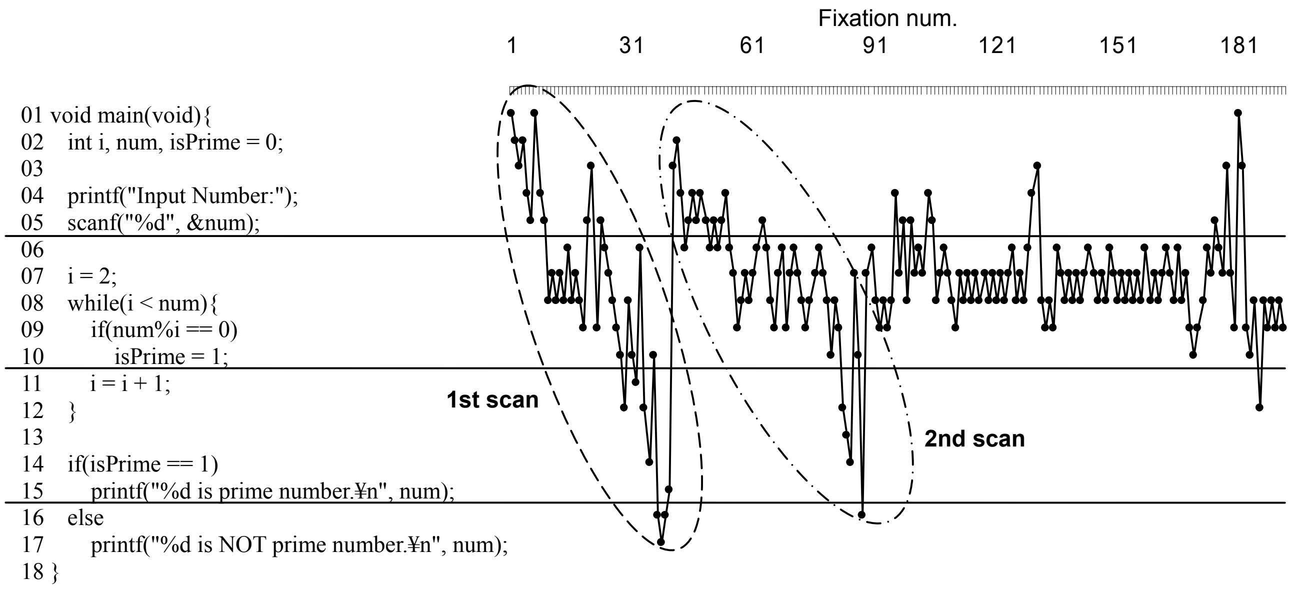
Program name	**Lines of code**	**Program specification**	**Defect injected in the source code**
`Sum-5`	12	The user inputs five integers. The program outputs the sum of these integers.	A variable accumulating the sum is not initialized.
`Accumulate`	20	The user inputs a non-negative integer *n*. The program outputs the sum of all integers from 1 to *n*.	A loop condition is mistaken. The condition must be (i <= n) but it is (i < n).
`Average-5`	16	The user inputs five integers. The program outputs the average of these.	An explicit type conversion from integer to double is forgotten, yielding a round margin in the average.
`Average-any`	22	The user inputs an arbitrary number of integers (up to 255) until zero is given. The program outputs the average of the given numbers.	The number of loops is wrong. The program always calculates the average of 255 numbers regardless of the number of integers actually entered.
`Swap`	23	The user inputs two numbers. The program swaps these numbers by using a function “swap()”. Then the program outputs the result.	The pointers are misused. As a result, the two numbers are not swapped.
`Prime`	18	The user inputs an integer *n*. The program checks whether *n* is a prime number or not.	Logic in a conditional expression is wrongly reversed, yielding an opposite verdict.

code of `Prime`. Using the result viewer extensively, we analyzed the eye movements of the individual subjects. As a result, a particular pattern of the eye movements was identified.

Scan Pattern

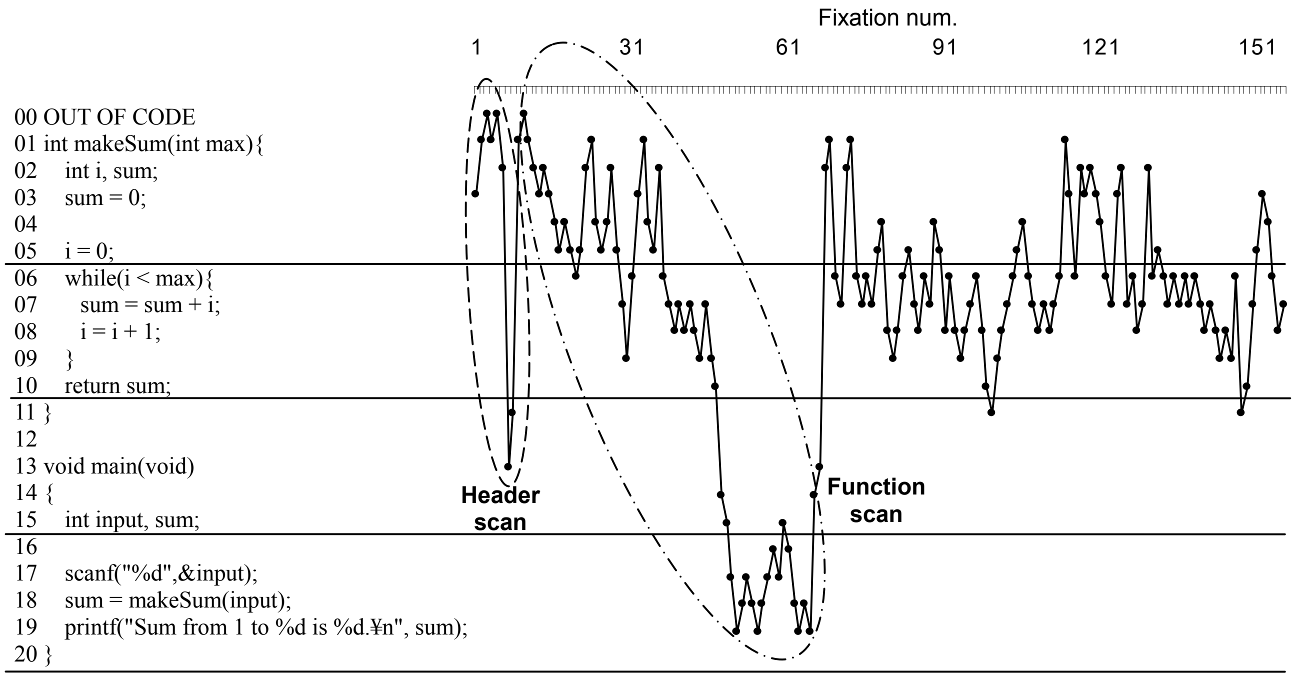
It was observed that the subjects were likely to first read the whole lines of the code from the top to the bottom briefly, and then to concentrate some particular portions. The statistics show that 72.8 percent of the code lines were watched in the first 30 percent of the review time. We call *scan* to represent this preliminary reading of the entire code.

Fig. 5 describes eye movements of two subjects where the *scan* patterns are well observed. The graphs depict the time sequence of code lines focused. As seen in Fig. 5 (a), Subject E scans the code twice, then concentrates the while loop block located middle of code. In Fig. 5 (b), it is seen that Subject C firstly locates the headers of two function declarations in lines 1 and 13. Then, the subject scan the two functions makeSum() and main() in this order.



```
01 void main(void){
02   int i, num, isPrime = 0;
03
04   printf("Input Number:");
05   scanf("%d", &num);
06
07   i = 2;
08   while(i < num){
09     if(num%i == 0)
10       isPrime = 1;
11     i = i + 1;
12   }
13
14   if(isPrime == 1)
15     printf("%d is prime number.¥n", num);
16   else
17     printf("%d is NOT prime number.¥n", num);
18 }
```

a) Subject E reviewing `Prime`



```
00 OUT OF CODE
01 int makeSum(int max){
02   int i, sum;
03   sum = 0;
04
05   i = 0;
06   while(i < max){
07     sum = sum + i;
08     i = i + 1;
09   }
10   return sum;
11 }
12
13 void main(void)
14 {
15   int input, sum;
16
17   scanf("%d",&input);
18   sum = makeSum(input);
19   printf("Sum from 1 to %d is %d.¥n", sum);
20 }
```

b) Subject C reviewing `Accumulate`

Fig. 5. Eye movements involving scan pattern

After the scan, he concentrates on the review of makeSum().

Characterizing Review Performance by Scan Pattern

It is reasonable to consider that the scan pattern reflects a cognitive action in code review; a reviewer firstly tries to understand the whole program structure. During the scan, a reviewer should identify some suspected portions where the defect is likely to be contained. Therefore, we consider that the quality of the scan should significantly influence the individual efficiency of the defect detection in the review.

For each review in the experiment, we measured *first scan time* and *defect detection time*. The first scan time is defined as the time spent from the beginning of the review until 80 percent of the total lines (except blank lines) are read. On the other hand, the defect detection time is the time taken for a reviewer to detect the injected defect. Assuming that the first scan time reflects the quality of the scan, we analyze the correlation among the first scan time and the defect detection time.

Fig. 6 depicts a scattered plot, representing the results of individual review with respect to the first scan time and the defect detection time. In the figure, the horizontal axis represents the first scan time, whereas the vertical axis represents the defect detection time. Each axis is normalized by the average. The figure shows a tendency that the first scan time less than the average yields the longer defect detection time. Specifically, the defect detection time increased up to 2.5 times of average detection time when the first scan time is less than 0.8. On the other hand, in the case that the scanning time is more than 0.8, the defect detection time is less than the average.

The experiment showed that the longer a reviewer scanned the code, the more efficiently the reviewer could find the defect in the code review. This observation can be interpreted as follows. A reviewer, who carefully scans the entire structure of the code, is able to identify many *candidates* of code lines containing defects during the scan. On the other hand, a reviewer with insufficient scan is likely to miss some critical code lines and stick to irrelevant lines involving no defect, which decreases the performance of the code review.

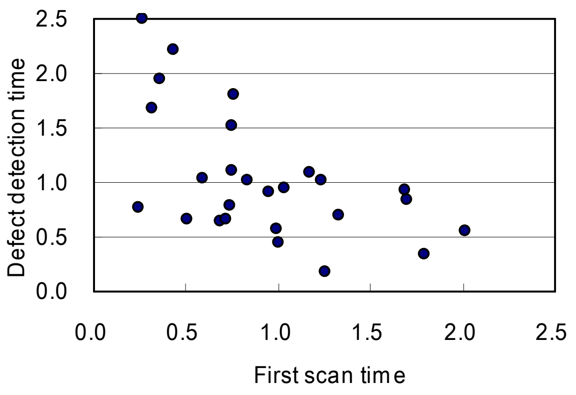


Fig. 6. Correlation between first scan and defect detection

5 Discussion

We here discuss other interesting findings, although their correlation with the review performance is not yet proven quantitatively.

5.1 Eye Movements and Reviewer's Thought

After the experiment, we conducted *two* kinds of interviews to investigate what the eye movements actually reflect.

In the first interview, for each subject we showed the source code and asked what the subject had been thinking in the code review. Most subjects commented general (or abstract) review policies, including the strategy of understanding the code and the flow of the review. Typical comments are summarized in the first column of Table 2.

In the second interview, we showed the eye movements with the result viewer as well as the source code, and asked the same questions. As a result, we were able to gather more detailed and code-specific comments. As shown in the second column of Table 2, each subject told reasons why he checked some particular lines carefully and why not for other lines. It seems that the record of the eye movements reminded the subjects of their thought well.

This fact indicates that the eye movements involve much information reflecting the reviewer's thought during the code review. Therefore, captured data of expert reviewers might be used for educational/training purposes.

5.2 Other Reading Patterns

In the experiment, we have found several interesting reading patterns other than the scan pattern. Due to the limited pages, we here present typical two patterns among them.

Retrace Declaration Pattern

When a reviewer reaches a code line where a variable is firstly used, within a short period the reviewer often looks back to the *declaration* line of the variable. We define this eye movement as *retrace declaration* pattern. Fig. 7 shows the eye movements involving the pattern. It can be observed that when the subject reaches the line 4 involving the first reference of variable *i*, he looks back the line 2 twice, where *i* is declared. The same patterns can be seen for variable *input* at line 6, and *sum* at line 7. Statistics show that the number of variables causing the retrace declaration pattern is 51.8 percent of the total number of variables. The retrace declaration pattern can be interpreted as a cognitive action that the reviewer reconfirms the *data type* of the variable.

Retrace Reference Pattern

This pattern is similar to the retrace declaration. When a reviewer reaches a code line where a variable is used, within a short

Table 2 Comments gathered in interviews

First interview (with source code only)	Second interview (with source code and eye movements)
• I reviewed the main function first, and then read another. • I felt something is wrong in the second while loop. • I simulated the program execution in mind assuming an input value. • I checked the while loop for a number of times.	• I did not check the conditional expression of loop. • I watched this variable declaration to see the initial value of the variable. • I did not mind to this output process. • I thought this input process was correct because it is written in a typical way. • I could not understand the variable initialized like this.

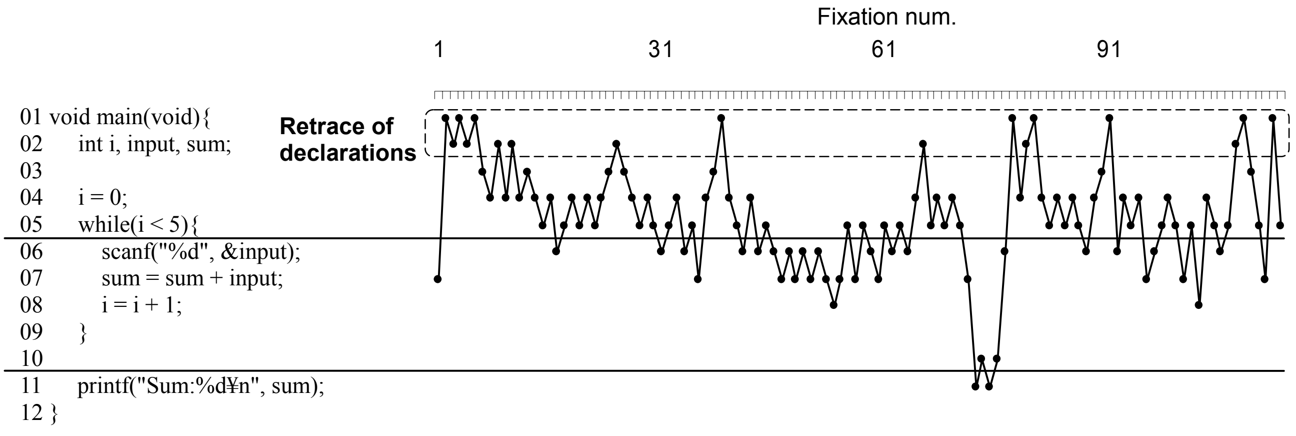


Fig. 7. Eye movements involving retrace declaration pattern (Subject A reviewing `Sum-5`)

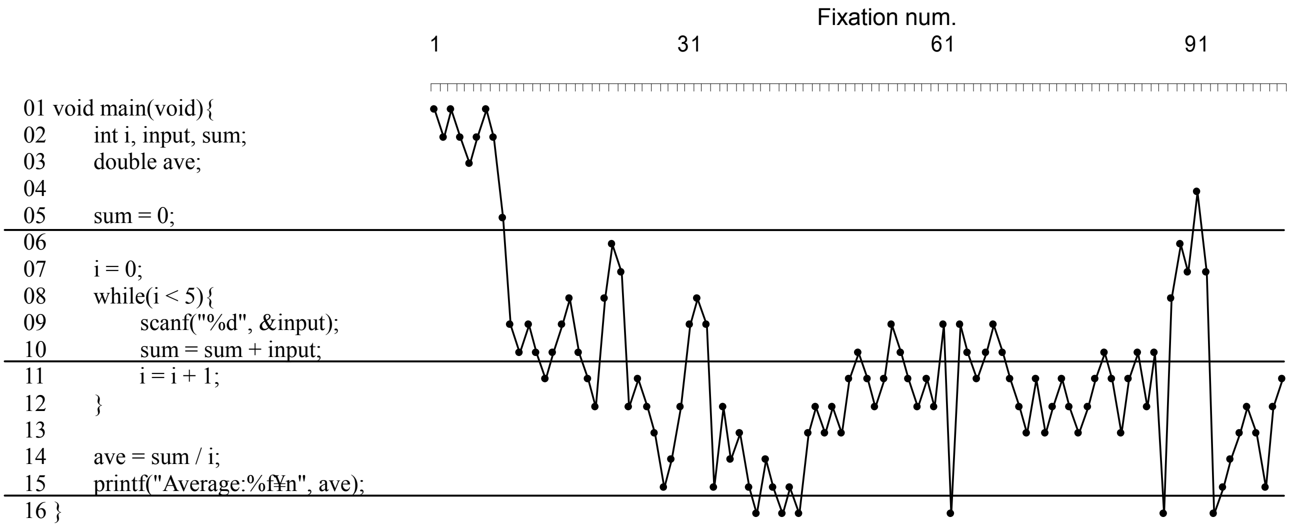


Fig. 8. Eye movements involving retrace reference pattern (Subject C reviewing `Average-5`)

period the reviewer often looks back to the lines where the variable has been *recently referred*. This pattern is defined as *retrace reference* pattern. Fig. 8 presents the eye movements containing this pattern. In the figure, when the subject reaches line 15 involving a variable *ave*, he looks back line 14 first where *ave* is recently referred. Then, since line 14 contains two variables sum and i, a further retrace pattern occurs back to the *while* block, reading the lines containing *sum* and *i*. The pattern can be interpreted as a cognitive action that the reviewer remember and recalculate the *value* of the variable.

It is reasonable to consider that the above reading patterns intuitively reflect cognitive aspects of a reviewer, and thus the patterns should affect the individual performance of code review. However, the experiment conducted in this paper did not show any significant correlation with the performance. Also in the subsequent interviews, we were not able to gather any specific comments from the subjects unfortunately. To clarify these patterns, we need more experiments with larger programs and more subjects. Further investigation of the other reading patterns is left as our future work.

6 Conclusion

In this paper, we have proposed to use the eye movements to analyze individual performance of source code review. We first developed an integrated measuring environment of eye movements in code review, including a software application Crescent. Then, using the environment, we conducted an experiment to characterize the review performance with eye movements. In the experiment, we found a particular reading pattern, called scan. Through the statistic analysis, it was shown that the reviewers taking sufficient time for scanning the code tend to detect defects efficiently.

Some topics for future research present themselves. We are planning to conduct experiments with more practical settings. Through more experiment we examine more reading patterns and their impacts to the individual review performance. It is also an interesting challenge to apply Crescent to peer review of other documents like requirements and specifications.

References

BASILI, V. R., GREEN, S., LAITENBERGER, O., LANUBILE, F., SHULL, F., SØRUMGÄRD, S., and ZELKOWITZ, M. V. 1996. The Empirical Investigation of Perspective-Based Reading, *Empirical Software Engineering: An International Journal*, 1, 2, 133-163.

BOEHM, B. W. 1981. *Software Engineering Economics*. Prentice

Hall.

BOJKO, A., and STEPHENSON, A. 2005. Supplementing Conventional Usability Measures with Eye Movement Data in Evaluating Visual Search Performance. In *Proceedings of the 11th International Conference on Human-Computer Interaction (HCII 2005).*

CIOLKOWSKI, M., LAITENBERGER, O., ROMBACH, D., SHULL, F., and PERRY, D. 2002. Software Inspection, Reviews & Walkthroughs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 641-642.

CROSBY, M., E., and STELOVSKY, J. 1990. How Do We Read Algorithms? A Case Study. *IEEE Computer*, 23, 1, 24-35.

FAGAN, M. E. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15, 3, 182-211.

FUSARO, P., LANUBILE, F., and VISAGGIO, G. 1997. A Replicated Experiment to Assess Requirements Inspection Techniques. *Empirical Software Engineering: An International Journal*, 2, 1, 39-57.

HALLING, M., BIFFL, S., GRECHENIG, T., and KOHLE, M. 2001. Using Reading Techniques to Focus Inspection Performance. In *Proceedings of 27th Euromicro Workshop Software Process and Product Improvement*, 248-257.

JACOB, R. J. K. 1995. Eye Tracking in Advanced Interface Design. *Virtual environments and advanced interface design*, Oxford University Press, 258-288.

KASARSKIS, P., STEHWIEN, J., HICHOX, J., ARETZ, A., and WICKENS C. 2001. Comparison of Expert and Novice Scan Behaviors during VFR Flight. In *Proceedings of the 11th International Symposium on Aviation Psychology,* http://www.aviation.uiuc.edu/UnitsHFD/conference/proced01.pdf

LANUBILE, F., and VISAGGIO, G. 2000. Evaluating Defect Detection Techniques for Software Requirements Inspections. *ISERN Technical Report*, 00, 08.

LAW, B., ATKINS, M. S., KIRKPATRICK, A. E., LOMAX, A. J., and MACKENZIE, C. L. 2004. Eye Gaze Patterns Differentiate Novice and Expert in a Virtual Laparoscopic Surgery Training Environment. In *Proceedings of ACM Symposium of Eye Tracking Research and Applications (ETRA)*, 41-48.

MILLER, J., WOOD, M., ROPER, M., and BROOKS, A. 1998. Further Experiences with Scenarios and Checklists. *Empirical Software Engineering: An International Journal*, 3, 3, 37-64.

NAKAMICHI, N., SAKAI, M., HU, J., SHIMA, K., NAKAMURA, M., and MATSUMOTO, K. 2003. Web-Tracer: Evaluating web usability with browsing history and eye movement. In *Proceedings of 10th International Conference on Human-Computer Interaction (HCI International 2003)*, 813-817.

PORTER, A., and VOTTA, L. 1998. Comparing Detection Methods for Software Requirements Inspection: A Replication Using Professional Subjects. *Empirical Software Engineering: An International Journal*, 3, 4, 355-380.

PORTER, A. A., VOTTA. L. G., and BASILI, V. R. 1995. Comparing Detection Methods for Software Requirements Inspection - A Replicated Experiment. *IEEE Transaction on Software Engineering*, 21, 6, 563-575.

SANDAHL, K., BLOMKVIST, O., KARLSONN, J., KRYSANDER, C., LINDVALL, M., and OHLSSON, N. 1998. An Extended Replication of an Experiment for Assessing Methods for Software Requirements Inspections. *Empirical Software Engineering: An International Journal*, 3, 4, 281-406.

SHULL, F. J. 1998. *Developing Techniques for Using Software Documents: A Series of Empirical Studies.* PhD thesis, Univ. of Maryland.

SHULL, F., RUS, I., and BASILI, V. 2000. How Perspective-Based Reading Can Improve Requirements Inspections. *IEEE Computer*, 33, 7, 73-79.

STEIN, R., and BRENNAN, S. E. 2004. Another Person's Eye Gaze as a Cue in Solving Programming Problems. In *Proceedings of the 6th International Conference on Multimodal Interface*, 9-15.

THELIN, T., ANDERSSON, C., RUNESON, P. and DZAMASHVILI-FOGELSTRÖM, N. 2004. A Replicated Experiment of Usage-Based and Checklist-Based Reading. In *Proceedings of 10th IEEE International Symposium on Software Metrics (METRICS'04)*, 246-256.

THELIN, T., RUNESON, P., and REGNELL, B. 2001. Usage-Based reading - An Experiment to Guide Reviewers with Use Cases. *Information and Software Technology*, 43, 15, 925-938.

THELIN, T., RUNESON, P., and WOHLIN, C. 2003. An Experimental Comparison of Usage–Based and Checklist-Based Reading. *IEEE Transaction on Software Engineering*, 29, 8, 687-704.

TORII, K., MATSUMOTO, K., NAKAKOJI, K. TAKADA, Y., TAKADA, S., and SHIMA, K. 1999. Ginger2: An Environment for Computer-Aided Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 25, 4, 474-492.

WEIGERS, K. 2002. *Peer Reviews in Software – A Practical Guide*. Addison-Wesley (in Japanese).

ZHAI, S., MORIMOTO, C., and IHDE, S. 1999. Manual and Gaze Input Cascaded (MAGIC) Pointing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 246-253.